Persistent

Core Java: Exception Handling and Assertions

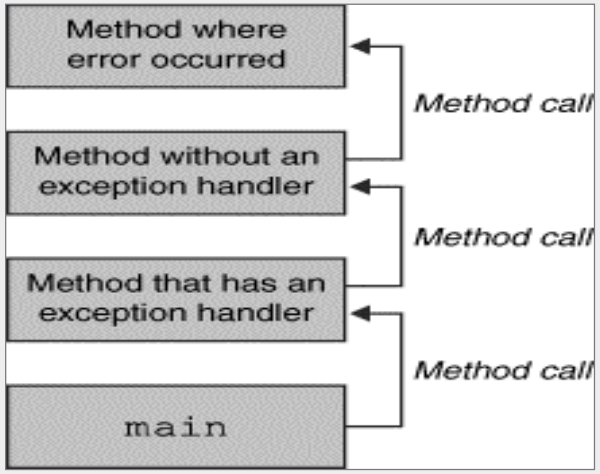
Persistent Interactive | Persistent University

Objectives :

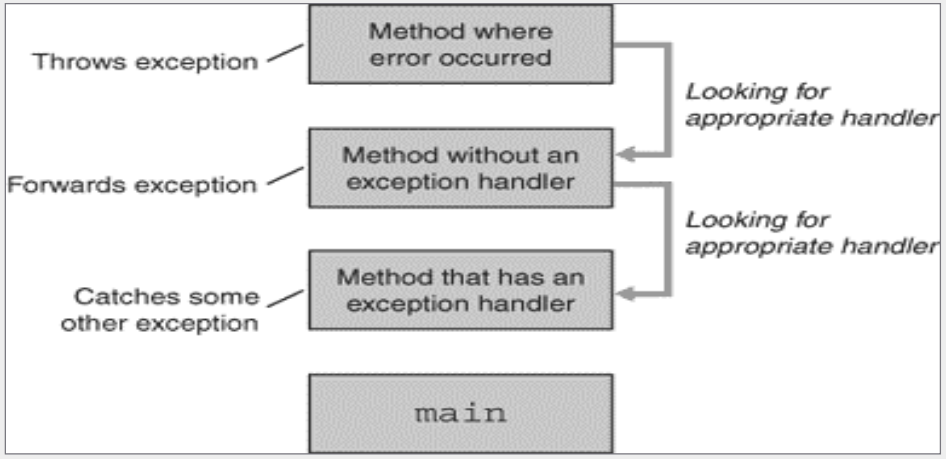
- In this module we will discuss about:
  - Stack-based Execution
  - Checked & Unchecked exceptions
  - Throw and Throws
  - Exception chaining
  - Catching multiple exception types
  - Re-throwing exceptions
  - Assertions

Stack based execution

- ordered list of methods that had been called to get to the method where an exception occurred.
- the runtime system searches the call stack for a method that contains a block of code that can handle the exception



Call Stack



Search for appropriate Handler

Persistent

Types of exceptions

- Checked exceptions

- Unchecked exceptions

Checked exceptions

- The compiler will confirm at compile time that the method includes code that might throw an exception.
- except for RuntimeException, Error, & their subclasses, all exceptions are called checked exceptions.
- must be included in a method's throws list.

Unchecked exceptions

- compiler does not check to see if a method handles or throws these exceptions.
- most general of these exceptions are subclasses of the standard type RuntimeException.
- need not be included in any method's throws list.

Custom exceptions

- Exceptions created by developers.

- These are as per the requirement of application.

Custom exceptions

- Steps to create custom exceptions:
- Create a class that extends Exception class.
- E.g. A developer needs to accept input from user as age and any value that is less than 18 is not a valid value.
- class AgeException extends Exception{}

Custom exceptions

- How to create custom exception

```
public class AgeException extends Exception{

        public AgeException(String msg){

                super(msg);
        }
}
```

Throw keyword

- The throw keyword is used to explicitly throw an exception

- The general form of throw is :

```
throw ThrowableInstance;
```

- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

Throw keyword

- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception. If it does find a match, control is transferred to that statement.
- If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

Throws keyword

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.

- You do this by including a throws clause in the method's declaration.

- A throws clause lists the types of exceptions that a method might throw.

Throw, Throw list

```
main() {
        method1();
    }

method1() {
        try {
                method2();
        } catch(Exception ex) {
                /* process exception */
        }
}

method2() throws Exception {
        if(some expression is not satisfied) {
                throw new Exception("failed");
        }
}
```

catch the exception that might originate from method2( )

method declares the type of exception it throws

method throws an exception but does not catch it

Exception chaining

- Chained exceptions allow you to throw a new exception, providing additional information without losing the original cause of the exception.

```
main()
 {
            try {

             method1();
             }
            catch(InterruptedException ex) {

            System.out.println(ex.getMessage());
            System.out.println(ex.getCause().getMessage());

 }    }
```

Exception chaining (Contd.)

method creates a new exception object, sets a cause for the new exception and lastly throws it

```
method1() throws InterruptedException {
        try {
        method2();
        } catch(IOException ex) {
        InterruptedException e;
        e = new InterruptedException(“interrupted”);
        e.initCause(ex);
        throw e;    }   }
method2() throws IOException {
        if(some expression is not satisfied) {
                throw new IOException(“failed”);
        }     }
```

Catching Multiple Exception Types

- It was required to write separate catch block for every exception getting thrown in try block.
- Now from JDK 1.7 one catch block can handle multiple exceptions.
- Each exception is separated by using pipe (|).

Catching Multiple Exception Types

```
public class ExceptionDemo {

public static void main(String[] args) {

   int a,b,c;   a = 10;  b = 2;
   try
   {
       c = a/b;
      System.out.println("c is: " + c);
       int arr[] = new int[3];
      String str = "hello";
      System.out.println(str.charAt(15));

  }
```

Catching Multiple Exception Types (Contd.)

```
catch(ArithmeticException e)
{
 e.printStackTrace();
}
catch(ArrayIndexOutOfBoundsException e)
{
 e.printStackTrace();
}
catch(StringIndexOutOfBoundsException e)
{
e.printStackTrace();
} } }
```

Catching Multiple Exception Types

```
public class SingleCatchBlock {

public static void main(String[] args) {

int b =10, x[] = { 10, 20, 30 };

try {
    int c = x[0]/b;
    System.out.println(c);
    String str = "hello";
System.out.println(str.charAt(15));

}
```

Catching Multiple Exception Types (Contd.)

```
catch(ArithmeticException |
 ArrayIndexOutOfBoundsException |
StringIndexOutOfBoundsException e)
{

 System.out.println(e);

}
}
}
```

Re-throwing an exception

- A method might sometimes re-throw an exception it caught, handled and received from another method. It might do so to first rollback, clean-up or do some kind of its own processing of the exception and then
- re-throw the exception to notify its own caller method of the exception.

Re-throwing an exception

```
main() {
        try {
                method1();
        } catch(Exception ex) { /* process */ }
}
method1() throws Exception {
        try {
                method2();
        } catch(Exception ex) {
                /* process exception */
                throw ex;
        }
}
method2() throws Exception {
        if(some expression is not satisfied) {
                throw new Exception("failed");
        }
}
```

Nitty-Gritty

- When necessary, a method can throw many exceptions:
- public void myMethod() throws IOException, OtherException
- When we throw an exception we use the throw keyword followed by an expression that creates an instance of the exception to be thrown:

```
throw new TheException();

TheException e = new TheException();

throw e;
```

Nitty-Gritty

- After a method throws an exception, the JVM begins the process of finding code to handle the error.
- The appropriate exception handler for a given exception is found by searching backwards through a chain of method calls, starting from the current method.
- If no handler is found, then the program terminates.
- Handlers are specified by the catch part of the try…catch construct.

try { ... }

catch (ExceptionType name) { ...}

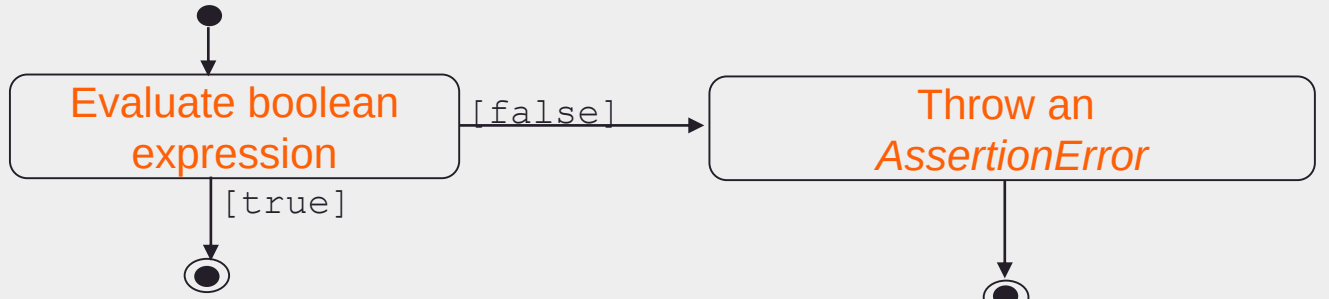
catch (ExceptionType name) { ... }

Assertions

- can be used to document and validate assumptions made about the state of the program at designated locations in the code.
- each assertion contains a boolean expression that is expected to be true when assertion is executed.
- if this assumption is false, the system throws a special assertion error.
- uses the exception handling mechanism to propagate error.
- can be enabled or disabled at runtime

assert statement and AssertionError class

Syntax	Interpretation
assert <boolean expression>;	if(<assertions enabled> && !<boolean expression>) throws new AssertionError();
assert <boolean expression>: <message expression>;	if(<assertions enabled> && !<boolean expression>) throws new AssertionError(<message expression>);



Normal execution continues

Execution is aborted and *AssertionError* propagated.

Persistent

Assertions……example

```
public class Speed {
public static void main(String[] args){
 Speed objRef = new Speed();
  double speed = objRef.calcSpeed(-12.0, 3.0);
                              // double
  speed = objRef.calcSpeed(12.0,-3.0);
                              // double
  speed = objRef.calcSpeed(12.0, 2.0);
                              // double
  speed = objRef.calcSpeed(-12.0, 0.0);

                System.out.println("Speed (km/h): " +speed);
                }
   /** Requires distance >=0.0 & time >0.0 */

  private double calcSpeed(double distance, double
time){
 assert distance >= 0.0;
 assert time >= 0.0: "Not a positive   value:" + time;
 double speed = distance / time;
 assert speed >= 0.0;
                              return speed;
}
```

Using assertions

- Assertion facility is a defensive mechanism
- Assertions should not be used to validate information supplied by a client
- Internal invariants:

```
int status = ref1.CompareTo(ref2)
if (status == 0) {
    ...
} else if (status >0) {
    ...
} else {
// status must be –ve
    ...
}
int status = ref1.CompareTo(ref2)
if (status ==0) {
    ...
} else if (status >0) {
    ...
} else {
    assert status <0: status;
    ...}
```

Summary

With this we have come to an end of our session, where we discussed about

- Exception handling framework overview
- Types of exceptions
- Custom exception
- Throw and throws clause

Appendix

References

Thank you

Reference Material : Websites & Blogs

- http://www.tutorialspoint.com/java/java_exceptions.htm
- http://tutorials.jenkov.com/java-exception-handling/index.html
- https://docs.oracle.com/javase/tutorial/essential/exceptions/

Reference Material : Books

- **Head First Java**
  - By: Kathy Sierra, Bert Bates
  - Publisher: O’Reilly Media, Inc.
- **Java Complete Reference**
  - By Herbert Schildt

Persistent

Thank you!

Persistent Interactive | Persistent University